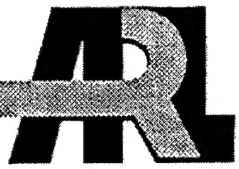
ARMY RESEARCH LABORATORY



Program Flow Graph Construction for Static Analysis of Explicitly Parallel Message-Passing Programs

by Dale R. Shires and Lori Pollock

ARL-TR-2370 November 2000

The findings in this report are not to be construed as an official Department of the Army position unless so designated by other authorized documents.

Citation of manufacturer's or trade names does not constitute an official endorsement or approval of the use thereof.

Destroy this report when it is no longer needed. Do not return it to the originator.

Army Research Laboratory
Aberdeen Proving Ground, MD 21005-5066

ARL-TR-2370 **November 2000**

Program Flow Graph Construction for Static Analysis of Explicitly Parallel Message-Passing Programs

Dale R. Shires
High Performance Computing Division, ARL

Lori Pollock
Computer and Information Sciences, University of Delaware

Abstract

In recent years, message-passing parallel codes have rallied around using the message passing interface (MPI). The parallelism in these codes is most often explicit; the developer must instrument the source code with calls to an optimized communications runtime library. MPI has been widely used for developing efficient and portable parallel programs, in particular for distributed memory multiprocessors and workstation/personal computer (PC) clusters, although its use in shared memory systems has been equally effective. This report presents algorithm for building a program flow graph representation of an MPI program. As an extension of the control flow graph representation of sequential codes, this representation provides a basis for important program analyses useful in software testing, debugging tools, and code optimization.

Acknowledgments

This research was performed while Dr. Pollock was on sabatical at the U.S. Army Research Laboratory's High Performance Computing Division. The author wishes to acknowledge the assistance given by Ms. Sara Sprenkle of Gettysburg College during the preparation of this manuscript.

INTENTIONALLY LEFT BLANK.

Table of Contents

INTENTIONALLY LEFT BLANK.

List of Figures

List of Tables

INTENTIONALLY LEFT BLANK.

1. Introduction

The message passing interface (MPI) is a library for writing distributed memory parallel programs that conform to a vendor-independent standard, thus enabling parallel programs that are portable to many platforms [1]. A significant number of large applications have been written in MPI, and it has been demonstrated that message passing programs written in MPI can be both efficient and portable to various parallel environments and architectures. However, MPI is often described as being analogous to assembly language programming due to the low-level details required of the programmer. The step from a sequential code to a correct, and not necessarily efficient, message passing parallel program is a challenging and time-consuming one. Few sophisticated program analysis, testing, or debugging tools exist to aid the programmer in this daunting task, as the design of these tools is complicated by concurrency, nondeterministic execution, data distribution, and communication.

Optimization of message passing programs has focused on aggregating communication, moving communication statements to hide communication latency by overlapping communication and computation, and reducing communication latency and unnecessary synchronization. However, techniques such as data flow analysis, classic optimization, data flow testing, and program slicing have not been addressed in the context of MPI programs.

Data flow analysis techniques for shared memory programs [2, 3], as well as data flow and dependence analysis for concurrent Ada programs [4] and distributed applications (i.e., those not of the form single program-multiple data [SPMD] [5, 6]), have been developed. Dynamic slicing methods for different models of concurrent programs have been developed for distributed programs with Ada-type rendezvous communication [7, 8], synchronous message passing distributed programs [9], and shared memory parallel programs [10]. Static slicing methods for concurrent programs [5, 11, 12] have focused on shared memory parallel programs with parallel sections and object-oriented features.

2. MPI Program Analysis

MPI programs present a different model of concurrent programming than these models. MPI programs are written in the SPMD style, in which each process executes the same program with unique data. Within these programs, special conditional statements based on the unique process identifiers allow for selectively executing various code segments. Although it is now possible in MPI-2 to create a multiple instruction-multiple data (MIMD) application by using a dynamic task creation feature, it is preferable to create a static SPMD MPI program, primarily for performance reasons. All processes are started as the program begins. Each process has its own local memory address space; there are no shared global variables among processes. All communication is performed through library calls to MPI routines.

This report describes efforts to develop a program representation for MPI programs that will enable static program analysis for software testing, debugging, and compiler optimization. All of these techniques require robust program understanding, achieved through good intertask data flow analysis and data dependency analysis. Calls to communication libraries in MPI explicity parallel code complicates all of these factors. However, these issues must be addressed to achieve the most optimized code possible. For example, automatic differentiation of functions containing message passing constructs is often less efficient than hand-coded versions. By providing a representation that will allow the compiler to perform a better analysis (dependence, control flow, data flow, etc.) in the presence of messages, the performance gap should shrink substantially [13].

Developing this representation for MPI programs introduces several challenges. First, the SPMD nature of the codes implies that the program representation for each process is not necessarily distinct. Rather, processes execute the same program, with segments to be executed by a subset of the processes designated by conditional statements. All processes execute the code that resides outside of these special conditionals. This behavior needs to be modeled correctly in the program representation and taken into account during static

program analysis. Second, programmers often exploit the rich set of collective communication routines in the MPI library, in addition to point-to-point communication. One way of handling programs with collective communication is translating them into a sequence of point-to-point communication calls for program analysis. However, one intended use of the representation is to display information about the program flow to the programmer through a graphical user interface (GUI); thus, the program representation should be presented to the programmer in terms of the original MPI program. Additionally, collective communication, such as `scatter` and `gather` operations, involve different sections of an array being partitioned or gathered to the various processes, respectively. It would be most useful to have the data flow information reflect this sectioning of the array.

The remainder of this report presents the results of a characterization study of a set of MPI programs that helped guide the design of these techniques, as well as an algorithm for constructing an MPI program flow graph.

3. Characterization Study

While the goal is to build a suite of "real-world" codes, finding stable MPI production codes is a common problem being addressed by consortiums and vendors. MPI usage was statically analyzed in the Numerical Aerospace Simulation (NAS) Parallel Benchmark suite and five other major codes listed in Table 1. The NAS codes include various kernel and application MPI benchmarks. ST3D from Washington University is a numerical relativity code that solves the full Einstein equations in three dimensions. CRUNCH3D from the Naval Research Laboratory addresses dissipative, compressible magnetohydrodynamics using three-dimensional (3-D) Fourier collocation. Znsflow from the U.S. Army Research Laboratory (ARL) is a computational fluid dynamics code that solves the unsteady Reynolds averaged Navier-Stokes equations and can be targeted to various projects of interest. OVERFLOW and BATSRUS come from NASA. OVERFLOW computes numerical solutions of the com-

Code	Source Lines	MPI Calls	Total Collective	Point-to-Point		
				Total	In Special Branches	Trivially Matched
NPB Block Tridiagonal	5432	54	6	24	0	12
NPB Multigrid	2438	41	7	13	0	12
NPB Scalar Pentadiagonal	4706	48	6	24	0	12
NPB 3-D FFT	1946	20	6	0	0	0
NPB LU Decomposition	5182	57	15	24	24	0
ST3D (Einstein)	15512	22	2	10	10	10
NRL CRUNCH3D	3207	48	11	6	6	6
ARL Znsflow	16744	58	17	19	0	19
NASA OVERFLOW	22017	457	40	374	≈318	≈318
NASA BATSRUS	16324	181	32	80	≈40	≈40

Table 1: Characteristics of MPI Usage in Parallel Programs

pressible Navier-Stokes equations by using a finite volume discretization in space and implicit time steps. BATSRUS is a magnetohydrodynamics code used for applications such as solar modeling. Some of the OVERFLOW and BATSRUS metrics are approximated based on a small sample from the code. This was necessary due to the code complexity, size, and lack of data flow analysis. A fully automated system with data flow analysis should provide more accurate results.

Several concluding observations were possible after a cursory examination of these programs. In all of the codes except one, the number of source lines related to MPI communications is less than 2% of the total number of lines of code (in most cases it is far less than 2%). The "MPI Calls" column in Table 1 gives a count of all MPI calls found in the code. The "Total Collective" column lists the number of MPI collective communications. Data pertaining to MPI point-to-point communication was further broken down. The "Total" column gives the number of sends and receives.

Special conditional statements (e.g., `if [myrank == 0]`) involving the process identifier are present, but not common. These statements are usually indicative of manager-worker style parallelism. The column "In Special Branches" shows the number of MPI sends and receives found in these branches. Many of these codes work on grid-based data, which seems

to naturally favor a data parallel approach to parallelism. As message passing codes, however, many of these programs rely upon initialization routines to compute arrays or scalars, such as `north`, `south`, etc., to hold information about neighboring processes and domains. Source and destination fields in the communication calls contain expressions using precomputed arrays, scalars, constants, and in some cases, the process identifier. MPI wildcards, such as `MPI_ANY_TAG` or `MPI_ANY_SOURCE`, are not common. Some codes use nested conditionals or loop constructs to further refine execution paths and interprocess communication.

Ultimately, while these programming styles make flow graph construction more difficult, they do not necessarily preclude it in most cases. Many of the scalars and arrays are defined with some reference to a unique process identifier. A static backward slice and variable substitution within the local process's flow graph can be used to reformulate these expressions in terms of the process identifier. Constant folding used in source and destination fields would also assist in the process.

Many point-to-point communication statements can be trivially matched. The number is given in the "Trivially Matched" column in Table 1. In these cases, a simple analysis of the communicator, type, and tag fields is enough to explicitly match communication statements.

4. MPI Program Flow Graph

Since each process in an MPI program has local space allocated for each of the declared variables in the program, and communication occurs only through matching MPI communication calls, data flow local to a given process between communication points in that process is not affected (as a side effect) by the data flow within other processes. The data flow within a given process is only affected by other processes at communication points. In point-to-point communication, a message sent by a particular send operation will be received by another process only through a receive operation executed by the other process. Specifi-

cally, a message can be received by a particular receive operation only if (1) it is addressed to the receiving process by the sender, (2) the send and receive have matching communicator fields, (3) the sender field of the receive is either `MPI_ANY_SOURCE`, or it matches the sender's process id, and (4) the tag fields of the send and receive match, or the tag field of the receive is `MPI_ANY_TAG`. In collective communication, all processes in the designated communicator are involved in the communication. Although multiple messages sent from one process to another process are guaranteed to arrive in the order they were sent, there are no assumptions made on the arrival order of messages from two different sources to the same destination. When a message receive specifies `MPI_ANY_SOURCE` as the expected sender, the originator of the message will be indeterminate at static analysis time; otherwise, the expected sender is specified, and communication is deterministic. Such indeterminacy is conservatively represented in this program representation.

A control flow graph (CFG) representation for a sequential program P is a directed graph $G = (N, E, S, e)$, where each node $n \in N$ represents a basic block of instructions, each edge $n \rightarrow m \in E$ represents a potential flow of control from node n to node m, and there is a unique start node s and a unique exit node e. A path in G is a sequence of nodes $(n_1, n_2, ...n_k)$, where $n_i \rightarrow n_{i+1}$ for all $1 \leq i \leq k$. It is assumed that every path in the CFG is a viable execution order of program P.

An *MPI-CFG* extends the CFG with communication edges and isolates each communication statement into its own separate basic block, represented by a single node in the graph. These nodes are called communication nodes.

While point-to-point communication can be easily represented by a single communication edge, collective communications have distinct semantics that result in different data flow across processes. For example, a broadcast will result in every process receiving the same value and storing it into the same local variable, whereas a scatter will result in each process receiving a subset of a set of values sent from the root process to distribute or partition the data stored in a single array among the processes. The representations of these communi-

cation statements were developed with the goal that each communication statement should have a unique representation that reflects its semantics.

Lastly, the control flow edges of the MPI-CFG are annotated with a value that reflects static information about the number, and possibly the process identifiers (if available) of the processes that could execute along that edge. The value will be one of the following four:

(1) $< c >$, indicating the known process id c of the only process that will execute that edge,

(2) $<$ *single* $>$, indicating that statically one can prove that only a single process will execute this edge, but one cannot determine the process id,

(3) $<$ *unknown* $>$, indicating that it could be one or more processes executing this edge, or

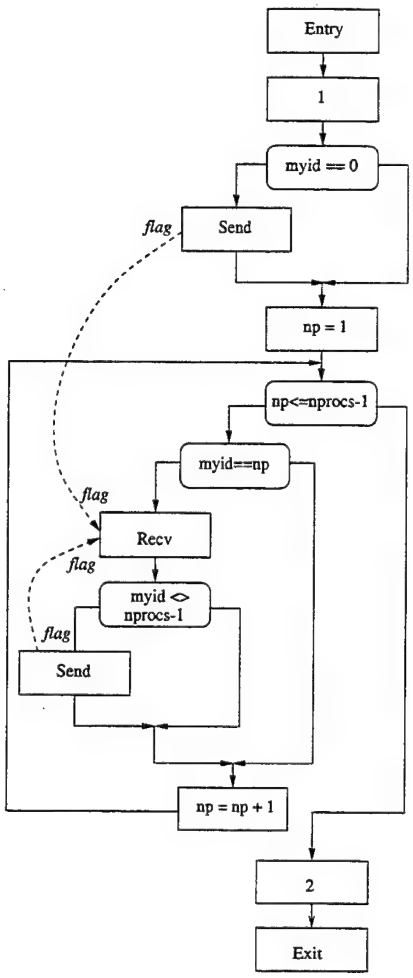
(4) $<$ *multiple* $>$, indicating that definitely more than one process will execute this code if there is more than one executing process.

A predicate annotation (e.g., myproc $<$ n) is also maintained if it is available and possible to identify. This information allows the communication edge addition step and other static program analyses to utilize the information about process ids.

Due to space constraints and the large size of most interesting MPI programs, it is preferable to define a *condensed* MPI-CFG as an MPI-CFG in which the nodes representing computation blocks between two communication nodes are collapsed into a single representative computation node. This structure is meant for presentation purposes only. Program analysis is to be performed over the MPI-CFG, not the condensed MPI-CFG. The MPI-CFG is illustrated in Figure 1. Figure 1(a) gives the code segment for an SPMD-style MPI program segment that performs a "cascading" style of communication, with processor 0 sending to 1, 1 to 2, etc. The MPI-CFG is shown in Figure 1(b). Control flow edges are indicated

```
   ...
   if (myid.eq.0) then
      call mpi_send(flag,1,MPI_INTEGER,1,tag,
&          MPI_COMM_WORLD,ierr)
   endif
   do np=1,nprocs-1
      if (myid.eq.np) then
         call mpi_recv(flag,1,MPI_INTEGER,np-1,
&             tag,MPI_COMM_WORLD,status,ierr)
         if (myid.ne.(nprocs-1)) then
            call mpi_send(flag,1,MPI_INTEGER,
&               np+1,tag,MPI_COMM_WORLD,ierr)
         endif
      endif
   enddo
    ...
```

(a) MPI Code Segment cascade.



(b) Corresponding MPI-CFG.

Figure 1: An Example MPI-CFG.

by solid lines, while communication edges are shown as dashed lines. Communication edges are labeled with the variables that are involved in the interprocess communication. The conditional $< myid == 0 >$ is an example of a special conditional statement indicating that the left branch is to be executed only by process 0, while the rest of the processes should execute the right branch.

5. Construction Algorithm

5.1 Basic Approach. The MPI-CFG construction algorithm is summarized in Figure 2. The first step is to create the underlying CFG by using a slight modification to the usual algorithm for CFG construction, which isolates communication statements as separate nodes. Each process's CFG is represented by some subgraph of this graph, where different processes typically have subgraphs that overlap one another. An initial pass of edge annotation, based on the relational operator of the special conditionals, will indicate program segments that are executed by one process vs. possibly multiple processes. Many parallel programmers program in the manager-worker style of programming, where the special conditionals `if [myrank == 0]` will often be an equality test against a constant. This information is used in the constant propagation phase. Traditional constant propagation can be applied to CFG representation of an SPMD program; however, it will be overly conservative in handling constants at join points from branches taken by different processes. More sophisticated constant propagation that recognizes constants with respect to particular processes would result in more precise information per process. Propagating constants helps to eliminate symbolic information in the parameters of communication statements, as well as the information known about the expressions in special conditionals.

The last step is to conservatively add communication edges. Because the same code segment may represent multiple processes, it is possible for a communication that occurs at runtime to have no associated communication edges, only a communication node. Communication edges are added according to the kind of communication, variables in particular fields of the communication call, any statically determined information about constants and the annotations on control flow edges, and the matching rules for communication statements. Sometimes the communication is ambiguous because of unknown values for variables, or wildcards in the source or tag fields. In these situations, an edge is added for any potential matching communication. In the MPI programs examined, there are very few communica-

```
Algorithm: MPI-CFG Construction.
Input: MPI program P.
Output: MPI-CFG representation of P.

begin
Treating MPI calls as regular function calls,
  Construct the CFG representation P-CFG of P;
Using the parameter of MPI_Comm_rank,
  Identify special conditionals that
    indicate separate process control flow;
Perform initial annotation of edges based on
  the expression operator in special
  conditionals;
Using annotations, perform modified constant
  propagation over  P-CFG;
Perform final annotation of edges using new
  information at special conditionals;
At each MPI communication statement,
  Use constants, CFG slices and MPI matching
    rules to identify potential matching
    communication;
Conservatively add communication edges to
  P-CFG;
end.
```

Figure 2: MPI-CFG Construction Algorithm.

tions that would cause additional edges to be added due to lack of information at analysis time.

The most challenging aspect of finding the potentially matching communication statements is identifying the source and destination processes. The source and destination fields of communication statements can be categorized as being (1) a constant, (2) an expression involving the process identifier, or (3) an expression not containing the process identifier. First, a traditional backward CFG slicing is performed (without communication edges) to reformulate expressions that are derived from the process identifier, but do not explicitly contain the process identifier. Then, in cases (1) and (3), the annotations on MPI-CFG edges are used to refine the set of potentially matching communications. In case (2), variable substitution is used in the expression functions of these fields to determine whether the

source and destination expressions of the receive and send operations, respectively, can be equal.

5.2 Extensions. Several enhanced control flow and data flow techniques are being considered. For example, several codes that were analyzed employ a programming style in which many `mpi_isend` statements are explicitly written. However, there is only one matching `mpi_irecv` statement, which is located in a function. This function is called repeatedly with the required parameters to match the various sends. Interprocedural analysis, or simply function inlining, will provide more information for static analysis.

Furthermore, approaches that may assist in providing more precise information for loop-nested communications are being investigated. Loop peeling, a technique useful in scalar replacement memory hierarchy optimizations, may prove beneficial [14]. The basic approach is to "peel" k iterations from the beginning of a loop and replace them with copies of the body and the associated increment and test code for the loop index. Where there are loop-nested communications in which the tag or source and destination fields are based on the loop index variable(s), peeling can be useful in restructuring the MPI-CFG to allow for better edge annotations. This technique should also be useful in removing communications edges that point into a loop body, thus simplifying the static slice.

6. Current Directions

The process of program flow graph construction is curently being implemented within the Stanford University Intermediate Format (SUIF) compiler infrastructure [15]. More precise constant propagation analysis is also being investigated. Studying various existing MPI codes revealed the need for a more robust constant folding technique. This should provide for better point-to-point communication matching and should allow for removing communication edges that are currently required to be conservative. Of particular interest is

the extension of the program dependence graph (PDG) representation for SPMD programs. The program dependence graph is a representation that succinctly represents both control and data flow in a program.

7. References

[1] Message Passing Interface Forum. "MPI: A Message-Passing Interface Standard." *International Journal of Supercomputer Applications*, vol. 8, no. 3-4, 1994.

[2] Grunwald, D., and H. Srinivasan. "Data Flow Equations for Explicitly Parallel Programs." *Fourth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pp. 159–168, 1993.

[3] Grunwald, D., and H. Srinivasan. "Efficient Computation of Precedence Information in Parallel Programs." *Languages and Compilers for Parallel Computing*, pp. 502–616. Springer-Verlag, 1993.

[4] Long, D., and L. A. Clarke. "Data Flow Analysis of Concurrent Systems That Use the Rendezvous Model of Synchronization." *Proceedings of the 4th Symposium on Testing, Analysis, and Verification*, pp. 21–35, 1991.

[5] Cheng, J. "Dependence Analysis of Parallel and Distributed Programs and Its Applications." *IEEE-CS International Conference on Advances in Parallel and Distributed Computing*, 1997.

[6] Cheung, S. C., and J. Kramer. "Tractable Dataflow Analysis for Distributed Systems." *IEEE Transactions on Software Engineering*, vol. 20, no. 8, pp. 579–593, August 1994.

[7] Korel, B., and R. Ferguson. "Dynamic Slicing of Distributed Programs." *Applied Mathematics and Computer Science*, 1992.

[8] Korel, B., and J. Laski. "Dynamic Program Slicing." *Information Processing Letters*, vol. 29, pp. 155–163, October 1988.

[9] Duesterwald, E., R. Gupta, and M. L. Soffa. "Distributed Slicing and Partial Reexecution for Distributed Programs." *Languages and Compilers for Parallel Computing*, pp. 497–511, 1992.

[10] Choi, J.-D., B. Miller, and R. Netzer. "Technique for Debugging Parallel Programs with Flowback Analysis." *ACM Transactions on Programming Languages and Systems*, vol. 13, no. 4, pp. 491–530, October 1991.

[11] Zhao, J., J. Cheng, and K. Ushijima. "Static Slicing of Concurrent Object-Oriented Programs." *Proceedings on IEEE-CS Twentieth Annual International Computer Software and Applications Conference*, pp. 312–320, 1996.

[12] Krinke, J. "Static Slicing of Threaded Programs." *Proceedings of the ACM SIGPLAN/SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, pp. 35–42. Montreal, Canada, 1998.

[13] Hovland, P. D. "Automatic Differentiation of Parallel Programs." Ph.D. thesis, University of Illinois at Urbana-Champaign, 1997.

[14] Muchnick, S. *Advanced Compiler Design and Implementation.* CA: Morgan Kaufmann Publishers, 1997.

[15] Stanford SUIF Compiler Group. *The SUIF Parallelizing Compiler Guide.* Stanford University, 1994.

NO. OF COPIES	ORGANIZATION
2	DEFENSE TECHNICAL INFORMATION CENTER DTIC DDA 8725 JOHN J KINGMAN RD STE 0944 FT BELVOIR VA 22060-6218
1	HQDA DAMO FDT 400 ARMY PENTAGON WASHINGTON DC 20310-0460
1	OSD OUSD(A&T)/ODDDR&E(R) R J TREW THE PENTAGON WASHINGTON DC 20301-7100
1	DPTY CG FOR RDA US ARMY MATERIEL CMD AMCRDA 5001 EISENHOWER AVE ALEXANDRIA VA 22333-0001
1	INST FOR ADVNCD TCHNLGY THE UNIV OF TEXAS AT AUSTIN PO BOX 202797 AUSTIN TX 78720-2797
1	DARPA B KASPAR 3701 N FAIRFAX DR ARLINGTON VA 22203-1714
1	NAVAL SURFACE WARFARE CTR CODE B07 J PENNELLA 17320 DAHLGREN RD BLDG 1470 RM 1101 DAHLGREN VA 22448-5100
1	US MILITARY ACADEMY MATH SCI CTR OF EXCELLENCE MADN MATH MAJ HUBER THAYER HALL WEST POINT NY 10996-1786

NO. OF COPIES	ORGANIZATION
1	DIRECTOR US ARMY RESEARCH LAB AMSRL D D R SMITH 2800 POWDER MILL RD ADELPHI MD 20783-1197
1	DIRECTOR US ARMY RESEARCH LAB AMSRL DD 2800 POWDER MILL RD ADELPHI MD 20783-1197
1	DIRECTOR US ARMY RESEARCH LAB AMSRL CI AI R (RECORDS MGMT) 2800 POWDER MILL RD ADELPHI MD 20783-1145
3	DIRECTOR US ARMY RESEARCH LAB AMSRL CI LL 2800 POWDER MILL RD ADELPHI MD 20783-1145
1	DIRECTOR US ARMY RESEARCH LAB AMSRL CI AP 2800 POWDER MILL RD ADELPHI MD 20783-1197
	ABERDEEN PROVING GROUND
4	DIR USARL AMSRL CI LP (BLDG 305)

INTENTIONALLY LEFT BLANK.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY *(Leave blank)*	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	November 2000	Final, May 1999 - September 2000

4. TITLE AND SUBTITLE
Program Flow Graph Construction for Static Analysis of Explicitly Parallel Message-Passing Programs

5. FUNDING NUMBERS
BCH02

6. AUTHOR(S)
Dale R. Shires and Lori Pollock*

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
U. S. Army Research Laboratory
ATTN: AMSRL-CI-HA
Aberdeen Proving Ground, MD 21005-5069

8. PERFORMING ORGANIZATION REPORT NUMBER
ARL-TR-2370

9. SPONSORING/MONITORING AGENCY NAMES(S) AND ADDRESS(ES)

10.SPONSORING/MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES
* University of Delaware, Newark, DE 19716

12a. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for public release; distribution is unlimited.

12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*

In recent years, message-passing parallel codes have rallied around using the message passing interface (MPI). The parallelism in these codes is most often explicit; the developer must instrument the source code with calls to an optimized communications runtime library. MPI has been widely used for developing efficient and portable parallel programs, in particular for distributed memory multiprocessors and workstation/personal computer (PC) clusters, although its use in shared memory systems has been equally effective. This report presents an algorithm for building a program flow graph representation of an MPI program. As an extension of the control flow graph representation of sequential codes, this representation provides a basis for important program analyses useful in software testing, debugging tools, and code optimization.

14. SUBJECT TERMS	15. NUMBER OF PAGES
MPI parallel programs, flow graph, optimization, program analysis, software tools	21
	16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
UNCLASSIFIED	UNCLASSIFIED	UNCLASSIFIED	UL

INTENTIONALLY LEFT BLANK.

USER EVALUATION SHEET/CHANGE OF ADDRESS

This Laboratory undertakes a continuing effort to improve the quality of the reports it publishes. Your comments/answers to the items/questions below will aid us in our efforts.

1. ARL Report Number/Author ARL-TR-2370 (Shires) Date of Report November 2000

2. Date Report Received _____

3. Does this report satisfy a need? (Comment on purpose, related project, or other area of interest for which the report will be used.) _____

4. Specifically, how is the report being used? (Information source, design data, procedure, source of ideas, etc.) _____

5. Has the information in this report led to any quantitative savings as far as man-hours or dollars saved, operating costs avoided, or efficiencies achieved, etc? If so, please elaborate. _____

6. General Comments. What do you think should be changed to improve future reports? (Indicate changes to organization, technical content, format, etc.) _____

CURRENT ADDRESS

_____
Organization

_____
Name E-mail Name

_____
Street or P.O. Box No.

_____
City, State, Zip Code

7. If indicating a Change of Address or Address Correction, please provide the Current or Correct address above and the Old or Incorrect address below.

OLD ADDRESS

_____
Organization

_____
Name

_____
Street or P.O. Box No.

_____
City, State, Zip Code

(Remove this sheet, fold as indicated, tape closed, and mail.)
(DO NOT STAPLE)

DEPARTMENT OF THE ARMY

OFFICIAL BUSINESS

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY MAIL
FIRST CLASS PERMIT NO 0001,APG,MD

POSTAGE WILL BE PAID BY ADDRESSEE

DIRECTOR
US ARMY RESEARCH LABORATORY
ATTN AMSRL CI HA
ABERDEEN PROVING GROUND MD 21005-5067